Operational Semantics for MSC'96

S. Mauw [a] and M.A. Reniers [a]

[a] *Department of Computing Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands.*

Abstract

Recently, the ITU-standardised specification language Message Sequence Chart has been extended with constructs for more complete and structured specifications. The new version of the language is called MSC'96. Currently, research is performed on the extension of the formal semantics towards a semantics for MSC'96.

In this article we aim at explaining the basic ideas behind the formal semantics. We give formal definitions of parts of the language, but most features are explained by informal examples and drawings.

It takes several steps in order to follow the path from an MSC drawing to its formal meaning. First, the drawing must be converted to a concrete textual representation. This conversion is already defined implicitly in Z.120. Next, this syntax is transformed into a formal expression over some process algebra signature. MSC constructs are replaced by appropriate process algebra operators. This transformation is compositional. The operational behaviour of the process algebraic expression can be studied, or the expression can be interpreted into some mathematical model and compared to the interpretation of some other MSC.

Key words: Message Sequence Chart, operational semantics, composition, standardisation.

1 Introduction

Recently, the specification language Message Sequence Chart (MSC) [16], which is standardised by the International Telecommunication Union (ITU), has been extended with constructs for more complete and structured specifications. The new version of the language is called MSC'96. Currently, research is performed on the extension of the formal semantics towards a semantics for MSC'96.

Ideally, the development of a language and its semantics should go hand in hand. There is little use in defining a fancy syntactic construction without a

precise understanding of its meaning. As was the case for the previous version of the MSC language, first the syntax and an incomplete and informal semantics were developed, while the construction of a formal semantics was deferred until after the acceptance of the language by the ITU-bodies. It is obvious that the a posteriori construction of a formal semantics will reveal many places in which the informal language description is ambiguous, underspecified, inconsistent or suboptimal. In [18] a number of such situations is described.

Nevertheless, there are also parts of MSC'96 that can be understood unambiguously. Of course, these are the parts of MSC'96 which are already covered by the old formal semantics. But, also the extension of the language with explicit and implicit operators for composing MSCs can be understood clearly. Most operators have already been studied in detail.

The purpose of this article is not to give a complete semantics definition of MSC'96. For this, we refer to the (upcoming) revision of Annex B to recommendation Z.120. We only aim at explaining the basic ideas behind the formal semantics. We give formal definitions of parts of the language, but most features are explained by informal examples and drawings. As the semantics are currently still under development, details may change. However, we expect that the basic ideas of the chosen approach remain stable.

The semantics proposed for standardisation are very much an extension of the previous formal semantics [22,15,20]. There are some differences, though. First, we have slightly changed the set of basic operators, such that it yields a smoother definition. Second, rather than giving a sound and complete process algebraic specification by means of axioms, we provide for an operational semantics based on process algebra expressions, only. The main reason is that we expect that a complete axiomatisation of all constructs involved is not feasible. This is mainly due to the extension towards infinite behaviour. The consequence of using an operational semantics is that we do not longer have the ability of equational reasoning. The equality of MSCs is now defined in a mathematical model, rather than by axioms.

Nevertheless, it is possible to define a sufficient number of sound equations which only fail in covering a few constructs completely. Although an important topic for research, we think that an incomplete set of axioms should not be part of the recommended semantics.

It takes several steps in order to follow the path from an MSC drawing to its formal meaning. First, the drawing must be converted to a concrete textual representation. This conversion is already defined implicitly in Z.120. Next, this syntax is transformed into a formal expression over some process algebra signature. MSC constructs are replaced by appropriate process algebra

operators. This transformation is compositional, in the sense that first the transformation of simple constructs can be determined, while the transformation of a complex construct is defined in terms of the transformations of the simple constructs. Finally, the operational behaviour of the process algebraic expression can be studied, or the expression can be interpreted into some mathematical model and compared to the interpretation of some other MSC.

In this article we focus on the definition of the process algebraic expressions and their operational meanings. This article is structured as follows. In Section 2 we give a short overview of the MSC'96 language. We make a distinction between *events*, constructs for *ordering* and constructs for *design*. In Section 3 we explain process algebraic expressions and operators and we discuss the transformation of an MSC into such an expression. In Section 4 we explain the operational semantics of some of the process algebraic operators needed.

2 MSC'96

The previous version of the MSC language, MSC'93 [14], was developed to express behavioural traces of distributed systems. It has primitives for denoting objects (*instances*) and for describing the relative order of messages exchanged between these objects. Furthermore, it has some more specific features, such as instance creation and timers. A specification consists of a series of MSCs, each describing a scenario. So-called *conditions* were used to guide the user through a specification.

Experience gained over the past few years, in which the use of the language increased, has shown that the language MSC'93 lacks the means to structure larger MSCs and to define the relation between simple MSCs. Therefore, most of the new features from MSC'96 deal with composition. Several important aspects have not been covered in MSC'96. These are amongst others time, data, probabilities and more specialised operators such as interrupts and disrupts. These are subject to further research.

In this section we give an overview of MSC'96, based on the viewpoint that all features fit in one of the three categories: *event*, *ordering*, *design*. We use these categories because the constructs within such a category are treated similarly in the definition of the semantics (see Section 3).

It is assumed that the reader has (at least) a basic knowledge of the language MSC'93. For a comprehensive treatment of the complete MSC'96 language we refer to [11]. An introduction to MSC'96 is [28].

2.1 Events

An *event* is the basic unit of observation. It models a part of the system's behaviour which is considered as one indivisible action. In MSC'96 we find the following events: *local actions*, message events (i.e. *sending* a message, *receiving* a message, a *lost message* and a *found message*), *creation* and *termination* of an instance and timer events (i.e. *set*, *reset* and *time-out*). The only new events are lost and found messages.

2.2 Ordering

An MSC is used to express the relative *ordering* of the events contained. The main source of order is the *instance*. In general, the events attached to an instance axis are in a strict sequential order. This order can be relaxed by the *coregion* construct. Events within a coregion may occur in any order. A basic assumption of the semantics of MSC is the requirement that in an MSC the sending of a message always precedes the corresponding reception. This ordering is referred to as the *message ordering*. These three means to describe ordering are already present in MSC'93.

MSC'96 extends MSC'93 with the *causal ordering* construct. By means of an arrow with the arrowhead in the middle events can be ordered causally. This feature allows for the description of ordering between events from different instances as well as the description of ordering on events from one instance. In combination with the coregion construct this makes it possible to express arbitrary partial orders on the events in a coregion.

2.3 Design

Several techniques have been introduced to support the modular *design* of specifications in a top-down or bottom-up manner. First, *High-level MSCs* (HMSC) [24] are used to indicate the relation between different smaller MSCs. In an HMSC one can express parallelism, sequencing, alternatives and recursion. HMSCs are the synthesis of the roadmap approach [13] and the operator approach [10]. As such they replace the informal use of roadmaps for overview specification.

With *MSC reference expressions* one is able to abstract from the actual contents of an MSC. In an MSC, references to other MSCs may be included, combined by several operators. A *substitution* mechanism is included which supports the reuse of modular specifications. Furthermore, the interface be-

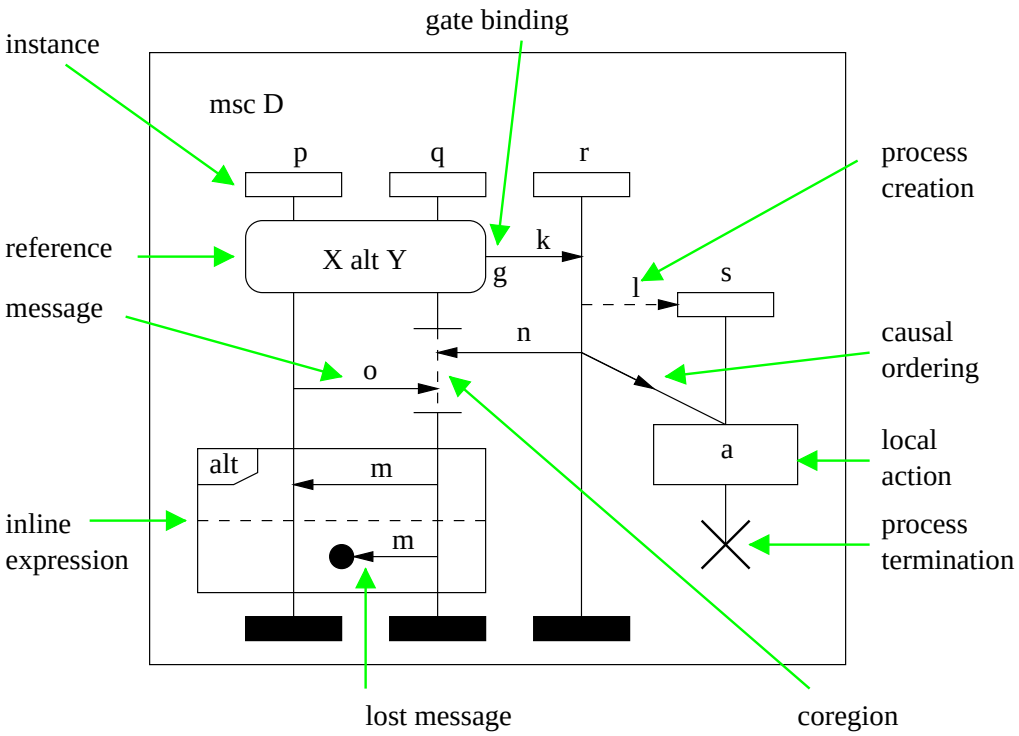

Fig. 1. Some MSC features.

tween an MSC reference and its environment may be defined by means of *gates*.

A similar construct is the *inline expression*. Operators are again used to relate several parts of an MSC. This is mainly intended to be used for small variations of a scenario.

In MSC reference expressions and inline expressions the following operators are at the disposal of the user: alternative ('alt'), sequential ('seq') and parallel composition ('par'), a family of loop operators ('loop'), and operators for describing optional behaviour ('opt') and exceptions ('exc').

Finally, we have *substructure references*, which stem from MSC'93. These allow one to aggregate the behaviour of a number of instances into one.

The graphical syntax of some of the abovementioned constructs is illustrated in the MSC depicted in Fig. 1.

3 Transformation

The first step in giving semantics to a language is the definition of a transformation from that language to a mathematical domain. In case of the opera-

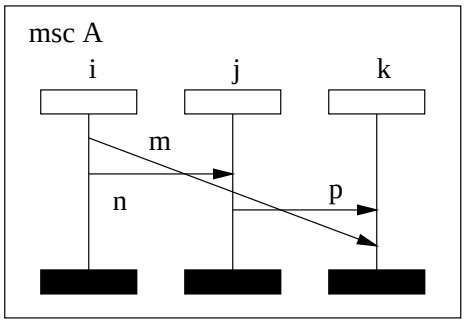

Fig. 2. Example MSC.

tional semantics for MSC we choose to transform MSCs into expressions over a signature that is introduced step by step in the remainder of this section. A signature describes a collection of constants and operator names, over which expressions may be formed. In this article the transformation appears to be defined on the graphical representation of MSCs. This is actually not the case, the textual syntax of MSC'96 is the starting point for the transformation. The one and only reason for this choice is that the graphical representation is by far not precise enough for a mathematical treatment. However, for the purpose of explaining the transformation we will, in this article, act as if the graphical syntax is used.

With the appearance of MSC'96 there are two main description styles for MSCs in the textual representation: the *instance-oriented* and the *event-oriented* description. With an instance-oriented description an MSC is given by describing all its instances, whereas with the event-oriented description an MSC is given by listing all building blocks (events, coregions, MSC references and inline expressions) contained in such a way that per instance the order of the building blocks respects the order of these building blocks in the graphical syntax from top to bottom.

The transformation presented here is defined on the event-oriented textual syntax of MSC'96. This is sufficient as every MSC can be expressed with this description style. Of course a formal definition of the transformation from an arbitrary textual description of an MSC to an event-oriented description should be provided. The choice for the event-oriented textual syntax is based on the observation that every MSC can be composed of building blocks by means of *vertical composition* only. In principle there are several ways to decompose an MSC into building blocks. It should be the case that different vertical decompositions of the same MSC result in 'equivalent' expressions.

To illustrate the decomposition of an MSC into its building blocks consider the MSC depicted in Fig. 2. The building blocks of this MSC are the message output and input events. In Fig. 3 the vertical decomposition of this MSC is indicated by means of dashed lines separating the building blocks. Note that we first had to rearrange the drawing. In the event-oriented representation of

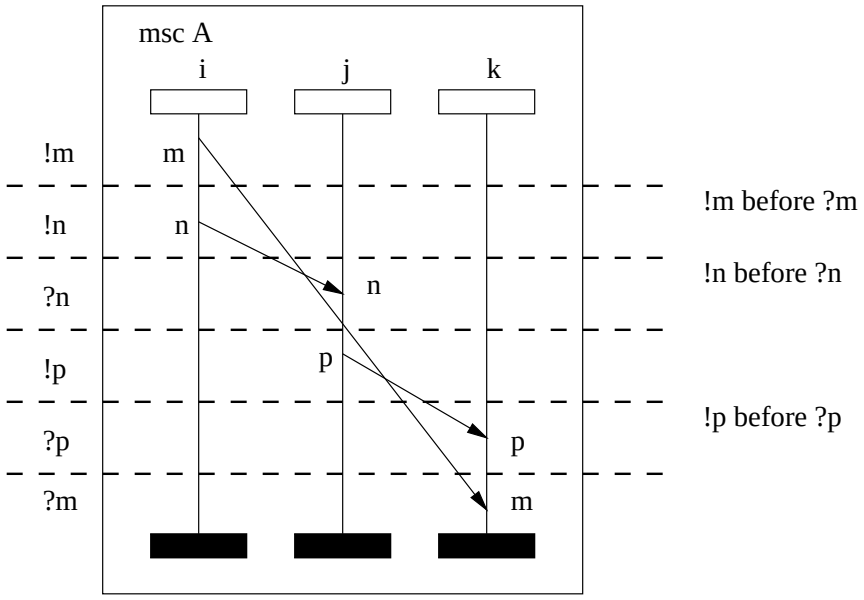

Fig. 3. Example MSC with separated events.

such an MSC the building blocks are already listed in a suitable order. Also for readability we duplicated the message identifiers in such a way that one can easily associate a message identifier with an event. For the moment ignore the annotations at the borders of the MSC in Fig. 3. They will be explained later.

The transformation is such that events are mapped to constants and that for each of the composition mechanisms offered by MSC'96 an operator is introduced into the signature to act as the semantical equivalent of the composition mechanism.

We describe the transformation for each of the classes used in the previous section to introduce the language MSC'96. It is impossible to give a treatment of the complete language MSC'96 in this article and, therefore, we focus on the compositionality of the transformation and the explanation of the basic ideas. Examples are added to increase the understanding. The MSC from Fig. 2 is used as a running example to explain the transformation described in this section.

3.1 Events

Any of the previously discussed events is represented by a constant in the signature. The set of all these constants is denoted by A and the constants are called *atomic actions*. Such an atomic action can have a number of arguments which give additional information such as the name of the event, the name of the instance it is attached to, etc.

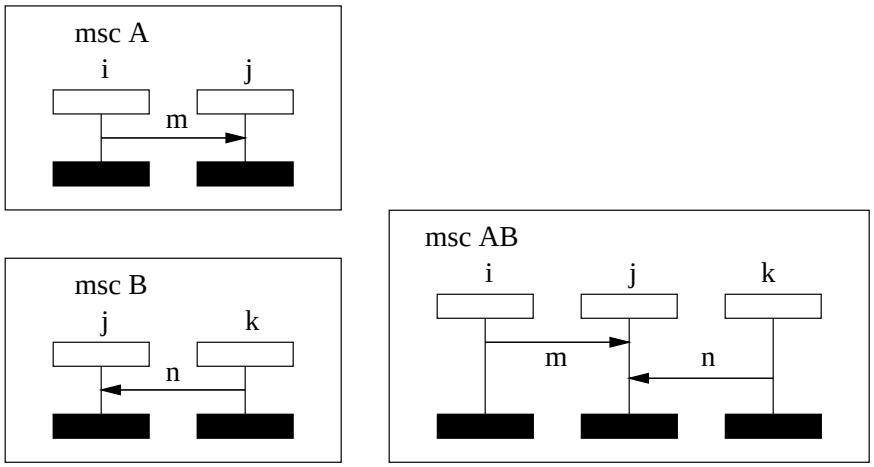

Fig. 4. Vertical composition.

For a local action with action name a which is attached to instance i we use the atomic action $action(i, a)$, the sending of a message m from instance i to instance j is described by $out(i, j, m)$ and the reception of this message by $in(i, j, m)$.

In the previously introduced example (see Fig. 2) all building blocks are events. On the left side of the MSC in Fig. 3 we indicate how these are represented in the semantics. Often, if no confusion can arise, we simply write $!m$ and $?m$ instead of $out(i, j, m)$ and $in(i, j, m)$. Similarly, $action(i, a)$ is represented by a. The reason for these shorthands is the readability of the expressions to follow.

3.2 Ordering

As described in Section 2.2 there are four ordering mechanisms in the MSC'96 language. The first of these, the ordering of events by placing them on an instance, is captured semantically by the *weak sequential composition* operator ($\circ$). This operator corresponds to the vertical composition of building blocks. If two building blocks are composed vertically the instances they have in common are linked in such a way that the events from such a common instance in the first building block precede the events on that instance in the second building block. A simple example of vertical composition is given in Fig. 4. The vertical composition of the MSCs A and B is denoted by $A \circ B$ and it is 'equivalent' with MSC AB. The vertical composition does not mean that all events from MSC A must be executed before events of MSC B are executed. For example the sending of message n can occur before the reception of message m.

The second way of ordering events is the coregion construct. In fact this construct is introduced to describe the absence of ordering. This means that the events in a coregion can be executed in any order. Semantically, this absence

of ordering is denoted by means of the *parallel composition* operator ($\|$). This operator corresponds to the *horizontal composition* of building blocks. If the building blocks that are composed have instances in common the behaviours of these are interleaved. A simple example to illustrate the representation of coregions is the following. Suppose that we have a coregion with events e_1, e_2, e_3 and e_4 and suppose that these events are semantically represented by the atomic actions a_1, a_2, a_3 and a_4. Then, semantically this coregion is represented by the expression $a_1 \| (a_2 \| (a_3 \| a_4))$.

In order to describe the causal orderings and the message orderings we need more machinery. For the message orderings we need to express that the message output precedes the message input. This can be achieved by using a state operator as was done in [22]. However, this approach does not generalise easily for the causal orderings. Therefore, we need a more general means to describe ordering. This is achieved by attributing the operators for the composition of building blocks with a set of ordering requirements. Such an ordering requirement is represented as $e_1 \mapsto e_2$ and should be read as: event e_1 must precede event e_2.

The attributed composition operators are called the *generalised weak sequential composition operator* ($\circ^S$) and the *generalised parallel composition* operator ($\|^S$). Usually we refer to these as vertical and horizontal composition. The weak sequential composition operator and the parallel composition operator are special cases of $\circ^S$ and $\|^S$ where $S = \varnothing$. Still, we prefer to denote $\circ^\varnothing$ by $\circ$ and $\|^\varnothing$ by $\|$. Often, we omit the curly brackets from the set S and simply list the ordering requirements separated by comma's. These attributes can be used to describe causal orderings, as well as message orderings. For a message m the message ordering would then be represented by the pair $!m \mapsto ?m$.

For the running example this means that it is described by the vertical composition of the building blocks and that at certain points additional ordering requirements are attributed to the vertical composition operator. Note that the orderings imposed by the instances are taken care of by the vertical composition operator on its own. After applying our transformation to MSC A from Fig. 3 the following expression results:

$$!m \circ^{!m \mapsto ?m} (!n \circ^{!n \mapsto ?n} (?n \circ (!p \circ^{!p \mapsto ?p} (?p \circ ?m)))).$$

This expression is obtained by considering the dashed lines in Fig. 3 one by one from top to bottom. With each dashed line an occurrence of $\circ^S$ is associated. If this line is crossed by a message arrow this results in an ordering requirement. Of course every such arrow should be considered only once.

The combination of building blocks is in MSC expressed explicitly by means of operators (inline expressions, MSC reference expressions) or by means of graphical constructions that can be reduced to operators (HMSCs). From a semantical point of view the three ways to combine building blocks can be described in one framework. In this article we do not consider substructure references. For a thorough treatment of those we refer to [23,15]. Also gates and substitutions are not treated in this article. Gates are difficult but not troublesome and substitution should be trivial.

Semantically, all we need for describing the composition mechanisms are the following operators: delayed choice ($\mp$) for 'alt', vertical composition ($\circ^S$) for 'seq', horizontal composition ($\|^S$) for 'par', several repetition operators to capture the 'loop' construct and recursion (for HMSC).

The delayed choice [2] is an operator for describing alternative scenarios in such a way that a choice between the alternatives is postponed for as long as possible. For example if both alternatives can execute an event a, then after the execution of a there are still two alternatives. If, on the other hand, only one of the alternatives can execute event a, then after the execution of a only one alternative remains. This is the alternative that executed a. The vertical and horizontal composition operators have been introduced already in the previous section. The constructs *optional* and *exception* (newly introduced in MSC'96) are easily captured as special cases of delayed choice and are therefore not considered in this article. The loop operators are based on bounded and unbounded versions of the iteration operator defined in [5]. Also these are not considered in this article. For an initial treatment of recursion we refer to [24].

We once more explain the transformation by means of an example. We start from the MSC given in Fig. 5. This example shows how to deal with complex building blocks such as coregions and inline expressions.

The first step is to make an alternative drawing in which a vertical composition of the MSC into building blocks can easily be indicated. A possible result of this transformation is shown in Fig. 6.

The next step is to associate an expression with the MSC as follows. First, with each of the building blocks an expression is associated. For the events this expression is merely a single atomic action and for the coregion we obtain $?n \,\|\, ?o$ as explained in the previous section. For the inline expression first an expression is given for each of the operands (two in this case): $!m1 \circ^{!m1 \mapsto ?m1} ?m1$ and $!m2 \circ^{!m2 \mapsto ?m2} ?m2$. Then these are combined by means of the the operator indicated in the inline expression. In this case this is the alternative composition operator 'alt' which is semantically represented by means of delayed

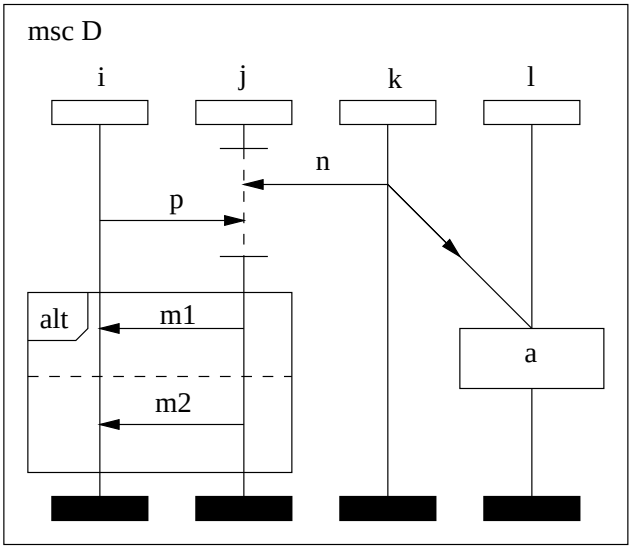

Fig. 5. Example MSC D.

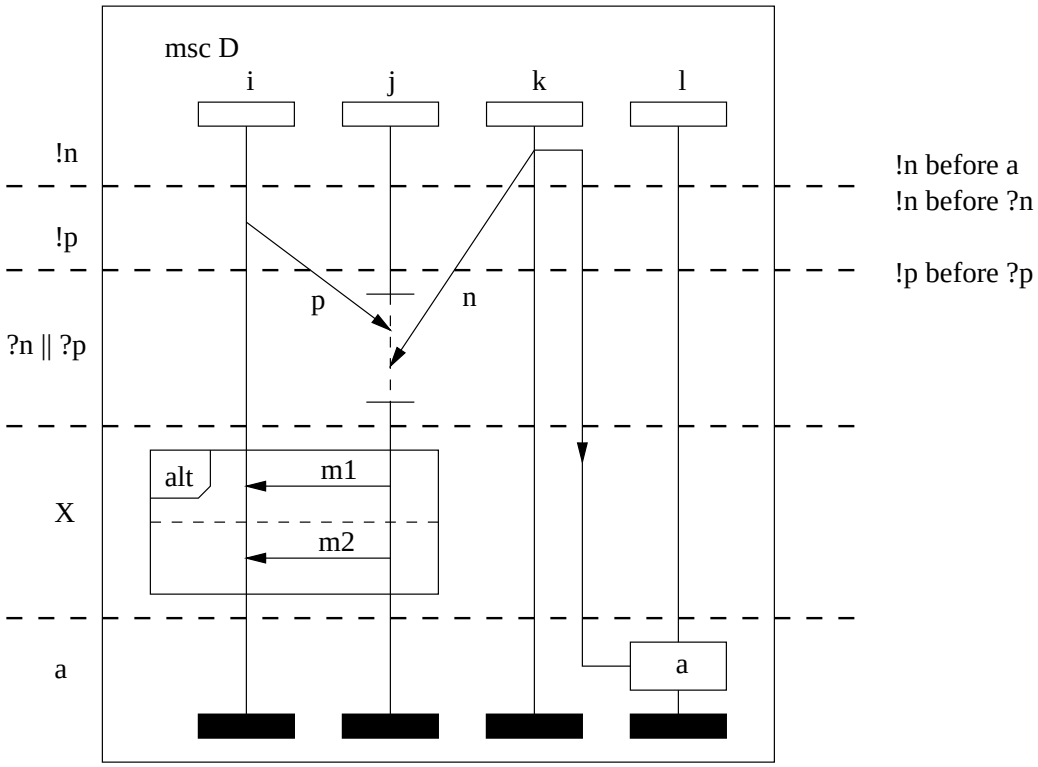

Fig. 6. Vertical decomposition of MSC D.

choice $\mp$. The result is the expression

$$(!m1 \circ^{!m1 \mapsto ?m1} ?m1) \mp (!m2 \circ^{!m2 \mapsto ?m2} ?m2).$$

The expressions associated with the building blocks are also displayed in Fig. 6 on the left of the MSC. The semantics of the inline expression is in the figure

abbreviated by X, which represents the above expression.

Then, the ordering requirements between the building blocks must be determined and taken into account. These are depicted in the figure on the right of the MSC. Note that in this step both the message orderings and the causal orderings are treated similarly. Finally, the expression representing the MSC is obtained as described before:

$$!n \circ^{!n \mapsto a, !n \mapsto ?n} (!p \circ^{!p \mapsto ?p} ((?n \parallel ?p) \circ (X \circ a))).$$

4 Operational semantics

4.1 What is an operational semantics?

In this section we introduce some terminology with respect to the mathematical framework that is used to define an operational semantics. Both terminology and notation are taken from [3]. The goal of an operational semantics is, given an expression denoting a process in a certain state, to describe all possible activities that can be performed by the process in that state and to describe the state of the process after such an activity. In the previous section we gave a transformation of an MSC into an expression. This expression represents the initial state of the MSC. The activities that are considered for the operational semantics of MSC'96 are the execution of an event and the termination of the MSC. Also the states resulting after such activities are described by means of expressions. If from a state s an event a can be performed and the resulting state is represented by the expression s', then this is usually denoted by the ternary relation $s \xrightarrow{a} s'$. If in a given state s the process is capable of terminating immediately and successfully, this is indicated by means of $s\downarrow$.

The predicate $-\downarrow \subseteq P$ is called the *termination predicate* as it indicates that a process has the possibility to terminate immediately and successfully. The set P denotes all expressions that can be built from the constants and operators in the signature. If we assume that all events are represented by atomic actions from the set A, the ternary relation $- \xrightarrow{-} - \subseteq P \times A \times P$ is called the *transition relation*. This predicate and these relations are defined by means of deduction rules (operational rules). A deduction rule is of the form $\frac{H}{C}$ where H is a set of premises and C is the conclusion. Each individual premise and the conclusion are of the form $s \xrightarrow{a} s'$ or $s\downarrow$ for arbitrary $s, s' \in P$ and $a \in A$. Such a deduction rule should be interpreted as follows: If all premises are true, the conclusion, by definition, also holds. A special kind of deduction rule appears if the set of premises is empty ($H = \varnothing$). Such a deduction rule is also called a deduction axiom and usually simply denoted by the conclusion C. An example

of a deduction axiom is deduction axiom (At 1):

$$a \xrightarrow{a} \varepsilon\,.$$

This deduction axiom expresses that a process that is in a state represented by the atomic action a can perform event a and thereby evolves into a state represented by the expression ε. This expression ε indicates a state in which no events can be performed but in which it is possible to terminate successfully and immediately. This is expressed by the deduction axiom (E 1):

$$\varepsilon\!\downarrow\ .$$

These are the only rules for expressions $a \in A$ and ε. The expression ε is used to denote an MSC without events.

Clearly the process a cannot terminate and the process ε cannot perform events. Note that these *negative* results are not explicitly defined. The following convention applies: If it is impossible to derive $s\!\downarrow$, then by definition not $s\!\downarrow$, which is denoted by $s \not\downarrow$. Similarly, if it is impossible to derive $s \xrightarrow{a} s'$, then by definition not $s \xrightarrow{a} s'$. This is usually denoted as $s \xnrightarrow{a} s'$. Such negative results can also be used in the set of premises, and then these are called *negative premises*. If we want to express that a process represented by the expression s can perform a transition labelled with a and we are not interested in the resulting state, this is denoted by $s \xrightarrow{a}$. Formally, it means that there exists a state s' such that $s \xrightarrow{a} s'$. Then $s \xnrightarrow{a}$ should be read as there does not exists a state s' such that $s \xrightarrow{a} s'$, or for all states s' we have $s \xnrightarrow{a} s'$. These abbreviations extend to the relation $- \overline{\cdots\!\rightarrow} -$ to be introduced in Section 4.4.

4.2 Delayed choice

The operational semantics associated with delayed choice by means of the deduction rules presented in Table 1 eminently illustrates the purposes of this operator. The deduction rules for $\xrightarrow{a}$ clearly express that $x \mp y$ can perform an a-transition thereby resolving the choice if exactly one of its operands can, and in the case that both operands can perform an a-transition the choice is not yet resolved.

The deduction rules from Table 1 are taken from [2] where the delayed choice operator was introduced in the setting of bisimulation semantics as a means for composing MSCs. The deduction rules (DC 1) and (DC 2) express that the alternative composition of two processes has the option to terminate if and only if at least one of the alternatives has this option. Thus, the process $action(i, a) \mp \varepsilon$ has an option to terminate as the second alternative has this option. On the contrary the process $action(i, a) \mp action(b, j)$ does not have an

Table 1
Deduction rules for delayed choice.

$$\frac{x\downarrow}{x \mp y\downarrow}(\text{DC 1}) \qquad \frac{y\downarrow}{x \mp y\downarrow}(\text{DC 2})$$

$$\frac{x \xrightarrow{a} x', y \not\xrightarrow{a}}{x \mp y \xrightarrow{a} x'}(\text{DC 3}) \qquad \frac{x \not\xrightarrow{a}, y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} y'}(\text{DC 4}) \qquad \frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} x' \mp y'}(\text{DC 5})$$

option to terminate as none of its alternatives can terminate. Please note that if we speak of termination we mean immediate termination, not termination after the execution of one or more events. The deduction rules (DC 3) and (DC 4) express that, in the situation that exactly one of the alternatives can execute an event a, the alternative composition can execute this event as well and that the execution of this event resolves the choice. Finally, deduction rule (DC 5) deals with the situation that both alternatives can execute an event a. It states that, in that case, the alternative composition can execute a and, moreover, that there remain two alternatives.

With the signature and deduction rules introduced so far it is hard to give interesting examples of how such an operational semantics can be exploited. But in order to already illustrate the machinery we give examples anyway.

As explained before, the delayed choice only resolves a choice if it has to. For example, if we consider the process $action(i,a) \mp action(i,b)$ making a choice between the alternatives cannot be avoided. Operationally this is seen as follows: By the deduction axiom (At 1) we obtain $action(i,a) \xrightarrow{action(i,a)} \varepsilon$. Since it is impossible to derive $action(i,b) \xrightarrow{action(i,a)}$ we obtain $action(i,b) \not\xrightarrow{action(i,a)}$. Combining these with the deduction rule (DC 3) we obtain $action(i,a) \mp action(i,b) \xrightarrow{action(i,a)} \varepsilon$ and the choice is resolved by the execution of $action(i,a)$. In contrast, for the process represented by $action(i,a) \mp action(i,a)$, we first obtain $action(i,a) \xrightarrow{action(i,a)} \varepsilon$, by deduction axiom (At 1), and therefore we cannot obtain $action(i,a) \not\xrightarrow{action(i,a)}$. Thus the deduction rule (DC 3) cannot be used this time. Instead the premises of deduction rule (DC 5) hold in this case. Thus we can conclude $action(i,a) \mp action(i,a) \xrightarrow{action(i,a)} \varepsilon \mp \varepsilon$. It is impossible to tell which $action(i,a)$ of the two alternatives was actually executed.

Table 2
Deduction rules for generalised parallel composition.

$S' = S \setminus \{e_1 \mapsto e_2 \in S \mid e_1 = a\}$ and $\pi_2(S) = \{e_2 \mid \exists_{e_1} e_1 \mapsto e_2 \in S\}$

$$\frac{x\downarrow, y\downarrow}{x \parallel^S y\downarrow}(\text{HC 1}) \qquad \frac{x \xrightarrow{a} x', y \not\xrightarrow{a}, a \notin \pi_2(S)}{x \parallel^S y \xrightarrow{a} x' \parallel^{S'} y}(\text{HC 2})$$

$$\frac{x \xrightarrow{a} x', y \xrightarrow{a} y', a \notin \pi_2(S)}{x \parallel^S y \xrightarrow{a} x' \parallel^{S'} y \mp x \parallel^{S'} y'}(\text{HC 3}) \qquad \frac{x \not\xrightarrow{a}, y \xrightarrow{a} y', a \notin \pi_2(S)}{x \parallel^S y \xrightarrow{a} x \parallel^{S'} y'}(\text{HC 4})$$

4.3 Generalised parallel composition

The generalised parallel composition operator is defined by the deduction rules in Table 2. It is labelled by a set S of pairs of atomic actions. This set specifies a number of ordering requirements on the execution of atomic actions. Before we explain the use of this set in more detail, we first explain the horizontal composition operator without considering this set. This turns out to be a special case, i.e., $S = \varnothing$. Instead of $\parallel^{\varnothing}$ we often write $\parallel$. This is in line with notation used in the process algebra ACP [6,4,3,29].

The horizontal composition of two processes is the interleaved execution of the events of the processes while maintaining the ordering of events as specified by the processes in isolation. The horizontal composition operator used in the setting of MSC, is a delayed version of the interleaving operators normally used. If both the left-hand and right-hand side of the horizontal composition can perform the same event, it is not visible which of the two is actually executed. In other words a delayed choice is made between the two occurrences. In this aspect the horizontal composition operator used for the semantics of MSC differs from the interleaving operators of ACP-style process algebras. Also the free merge operator (and weak sequencing operator) used in [24] does not make a delayed choice between alternatives.

A simple example is the process $action(i, a) \parallel action(i, b)$. This process is capable of performing an event $action(i, a)$ and thereby it evolves into the process $\varepsilon \parallel action(i, b)$. But it is also possible for this process to perform $action(i, b)$ and then the process $action(i, a) \parallel \varepsilon$ remains. An example illustrating the delayed nature of the horizontal composition operator is the process represented by $action(i, a) \parallel action(i, a)$. It can perform the following sequence of transitions:

$$action(i,a) \parallel action(i,a) \stackrel{action(i,a)}{\rightarrow} \varepsilon \parallel action(i,a) \mp action(i,a) \parallel \varepsilon$$
$$\stackrel{action(i,a)}{\rightarrow} \varepsilon \parallel \varepsilon \mp \varepsilon \parallel \varepsilon\downarrow \ .$$

Next, the way the set S is used is explained. The set S contains pairs of events $e_1 \mapsto e_2$. Such a pair describes a requirement on the order in which events from the operands may be executed. In this particular case the pair should be read as: event e_2 can only be executed after event e_1 has been executed. This is precisely what is expressed in the deduction rules. Also observe that the set S is updated after the execution of every event. This update is explained later.

Consider the process $in(i,j,m) \parallel^{out(i,j,m) \mapsto in(i,j,m)} out(i,j,m)$. If we do not consider the requirement $out(i,j,m) \mapsto in(i,j,m)$ it would be possible to execute the events $in(i,j,m)$ and $out(i,j,m)$ in any order. However, the presence of the requirement blocks the execution of $in(i,j,m)$ as long as $out(i,j,m)$ has not been executed. Thus, the only possible sequence of transitions is:

$$in(i,j,m) \parallel^{out(i,j,m) \mapsto in(i,j,m)} out(i,j,m) \stackrel{out(i,j,m)}{\rightarrow} in(i,j,m) \parallel \varepsilon \stackrel{in(i,j,m)}{\rightarrow} \varepsilon \parallel \varepsilon\downarrow \ .$$

Next, we explain the deduction rules from Table 2 in more detail. Rule (HC 1) expresses that $x \parallel^S y$ has the possibility to terminate if both x and y have. Rules (HC 2) up to (HC 4) define which transitions the expression $x \parallel^S y$ is allowed to make. These rules have the premise $a \notin \pi_2(S)$. This means that the event a is allowed to occur. Namely, set S contains pairs of events for which an ordering is fixed. If, e.g., $e_1 \mapsto e_2$ is in S, this means that e_1 should occur before e_2. After execution of e_1 the entry $e_1 \mapsto e_2$ is removed from S. Therefore, only events are blocked which occur at the right-hand side of an entry $e_1 \mapsto e_2$ in S. These right-hand sides are selected by the operation $\pi_2(S)$. This also explains why the set S in the conclusions of the rules is transformed into S'. Namely, if an a is executed, all requirements $a \mapsto e_2$ are trivially fulfilled and may thus be removed from the set S.

Knowing this, we can make a distinction between three cases. First, x can execute an a and y cannot ((HC 2)). In this case, $x \parallel^S y$ may perform this a and evolve into $x' \parallel^{S'} y$. The case that x cannot execute an a and y can is symmetrical ((HC 4)). Finally, if both x and y can execute an a ((HC 3)), we need a delayed choice to express that execution of this a does not enforce a choice between x and y. The executed a may be either due to x or due to y.

In this section we introduce the generalised weak sequential composition operator $\circ^S$. It can be thought of as the vertical composition of building blocks. The operator $\circ$, i.e., $\circ^{\varnothing}$, is based on the weak sequential composition operator of [27] and the interworking sequencing operator of [25]. It has a behaviour that is similar to the behaviour of the horizontal composition operator, but additionally it maintains the ordering of events from instances that the building blocks have in common.

In MSC every event is associated with an instance on which it is defined. In the operational semantics this is incorporated by assuming a mapping $\ell : A \to I$, where I represents the set of all instance names, which associates with every event $a \in A$ the name of the instance it is defined on. This mapping is easily defined on the atomic actions from the set A as these represent the events with the instance to which the event is attached as one of the parameters. For example $\ell(out(i,j,m)) = i$, $\ell(in(i,j,m)) = j$ and $\ell(action(i,a)) = i$. The mapping ℓ is used extensively in the operational description of vertical composition.

Besides the transition relation already present in the operational semantics we now also introduce a relation $- \cdots\!\!\bar{\to} - \subseteq P \times A \times P$. This relation is called the *permission relation*. If $s \overset{a}{\cdots\!\!\to} s'$ this means that in a situation where s is vertically composed with t ($s \circ t$) and t can perform event a ($t \overset{a}{\to}$), s allows the execution of a by t ($s \circ t \overset{a}{\to}$). If on the other hand $s \not\cdots\!\!\overset{a}{\to}$ this means that in the same situation the vertical composition of s and t cannot perform event a from t.

However, there is a complication with respect to alternatives. Suppose that we have an MSC that describes two alternatives: the first is the execution of local action a on instance i and the second alternative is the execution of local action b on instance j. Suppose furthermore that we wish to compose this MSC vertically with an MSC which can only execute local action c from instance i. The expression representing this vertical composition is given by

$$(action(i,a) \mp action(j,b)) \circ action(i,c).$$

Let us try to find out which transitions can be performed in this case. As before it is possible to perform the local action a or the local action b:

$$(action(i,a) \mp action(j,b)) \circ action(i,c) \overset{action(i,a)}{\to} \varepsilon \circ action(i,c)\,,$$
$$(action(i,a) \mp action(j,b)) \circ action(i,c) \overset{action(j,b)}{\to} \varepsilon \circ action(i,c)\,.$$

After this transition in both cases the choice has been made and clearly the

Table 3
Deduction rules for the permission relation.

$$\varepsilon \overset{a}{\cdots\!\!\rightarrow} \varepsilon \text{ (E 2)} \qquad \frac{\ell(a) \neq \ell(b)}{b \overset{a}{\cdots\!\!\rightarrow} b}\text{(At 2)} \qquad \frac{x \overset{a}{\cdots\!\!\rightarrow} x', y \not\overset{a}{\cdots\!\!\rightarrow}}{x \mp y \overset{a}{\cdots\!\!\rightarrow} x'}\text{(DC 6)}$$

$$\frac{x \not\overset{a}{\cdots\!\!\rightarrow}, y \overset{a}{\cdots\!\!\rightarrow} y'}{x \mp y \overset{a}{\cdots\!\!\rightarrow} y'}\text{(DC 7)} \qquad \frac{x \overset{a}{\cdots\!\!\rightarrow} x', y \overset{a}{\cdots\!\!\rightarrow} y'}{x \mp y \overset{a}{\cdots\!\!\rightarrow} x' \mp y'}\text{(DC 8)}$$

$$\frac{x \overset{a}{\cdots\!\!\rightarrow} x', y \overset{a}{\cdots\!\!\rightarrow} y'}{x \parallel^S y \overset{a}{\cdots\!\!\rightarrow} x' \parallel^S y'}\text{(HC 5)} \qquad \frac{x \overset{a}{\cdots\!\!\rightarrow} x', y \overset{a}{\cdots\!\!\rightarrow} y'}{x \circ^S y \overset{a}{\cdots\!\!\rightarrow} x' \circ^S y'}\text{(VC 1)}$$

next transition that can be performed is the execution of local action c. However, there is also the possibility that local action c appears even before the choice has been made. But then we know that local action a cannot be executed, because it should precede local action c. Therefore, the state after the execution of local action c, should not have the possibility to execute local action a anymore:

$$(action(i,a) \mp action(j,b)) \circ action(i,c) \overset{action(i,c)}{\rightarrow} action(j,b) \circ \varepsilon.$$

Suppose that we want to determine if an action a is allowed to precede a process x, i.e., we want to determine $x \overset{a}{\cdots\!\!\rightarrow}$. Then we use the deduction rules from Table 3 which are defined on the structure of the expressions. The empty process can be preceded by event a; there is no reason not to allow a to be executed. This is expressed by deduction rule (E 2). Similarly, an event b can be preceded by event a if and only if these events are defined on different instances ($\ell(a) \neq \ell(b)$), see deduction rule (At 2). The deduction rules (DC 6), (DC 7) and (DC 8) express that an event a is allowed to precede a process consisting of two alternatives if and only if at least one of the alternatives allows a to precede. For both the horizontal and the vertical composition operator the following holds. Event a can precede the composed process if and only if it can precede both operands ((HC 5) and (VC 1)). Please note that the operational rules also deal with the resolution of choices.

The operational rules for vertical composition in Table 4 are similar to the operational rules for horizontal composition. The main difference is that an event from the right operand can only be executed if the left operand allows the execution of that event. Also note that in case the left operand allows the execution of an event a from the right operand it is possible that alternatives disappear from the left operand.

Table 4
Deduction rules for generalised weak sequential composition.

$S' = S \setminus \{e_1 \mapsto e_2 \in S \mid e_1 = a\}$ and $\pi_2(S) = \{e_2 \mid \exists_{e_1} e_1 \mapsto e_2 \in S\}$

$$\frac{x\downarrow, y\downarrow}{x \circ^S y\downarrow}(\text{VC 2}) \qquad \frac{x \xrightarrow{a} x', y \overset{a}{\nrightarrow} \vee x \not\cdot^{a}\!\!\rightarrow, a \notin \pi_2(S)}{x \circ^S y \xrightarrow{a} x' \circ^{S'} y}(\text{VC 3})$$

$$\frac{x \xrightarrow{a} x', x \cdots^{a}\!\!\rightarrow x'', y \xrightarrow{a} y', a \notin \pi_2(S)}{x \circ^S y \xrightarrow{a} x' \circ^{S'} y \mp x'' \circ^{S'} y'}(\text{VC 4})$$

$$\frac{x \overset{a}{\nrightarrow}, x \cdots^{a}\!\!\rightarrow x', y \xrightarrow{a} y', a \notin \pi_2(S)}{x \circ^S y \xrightarrow{a} x' \circ^{S'} y'}(\text{VC 5})$$

Deduction rule (VC 2) expresses that the vertical composition of two processes can terminate if each of them can terminate. The other deduction rules deal with the possible transitions of the vertical composition of two processes. As the vertical composition operator deals with the ordering requirements in the same way as the horizontal composition operator this aspect of the deduction rules is not explained again.

In the case that x can perform event a ($x \xrightarrow{a}$) and either y cannot perform event a ($y \overset{a}{\nrightarrow}$) or x does not allow the execution of event a by y ($x \not\cdot^{a}\!\!\rightarrow$), only the execution of event a by x can take place. This is expressed by deduction rule (VC 3).

In the case that x can perform event a ($x \xrightarrow{a}$), y can perform event a ($y \xrightarrow{a}$) and x permits the execution of a by y, a delayed choice of the individual occurrences of event a results. This is expressed by deduction rule (VC 4).

In the case that x cannot perform event a ($x \overset{a}{\nrightarrow}$), y can perform event a ($y \xrightarrow{a}$) and x permits the execution of a by y ($x \cdots^{a}\!\!\rightarrow$), there is only the possibility that event a is executed due to y. This is expressed by deduction rule (VC 5).

The only case that is not discussed yet is the case that x cannot execute a and either x does not permit y to execute an event a or y cannot execute an event a. In this case it is impossible to execute a at all. This is expressed implicitly by the fact that there is no deduction rule for which the premises hold in this situation.

The operational rules for the vertical composition operator are illustrated by means of some examples. First, consider the MSCs from Fig. 7. The MSCs A and B have no instances in common. Semantically, they are represented by

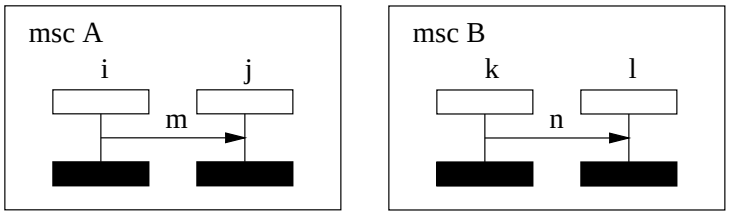

Fig. 7. Vertical composition with disjoint instances.

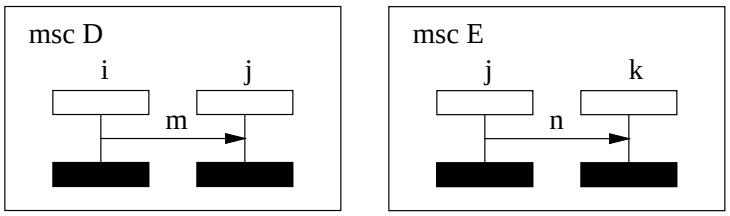

Fig. 8. Vertical composition with instances in common.

the following expressions, denoted by A and B for easy reference:

$$A = out(i,j,m) \circ^{out(i,j,m) \mapsto in(i,j,m)} in(i,j,m)\,,$$
$$B = out(k,l,n) \circ^{out(k,l,n) \mapsto in(k,l,n)} in(k,l,n)\,.$$

The vertical composition of MSC A and MSC B is then denoted by $A \circ B$. From the deduction rules we obtain that this process can perform the following transitions:

$$A \circ B \stackrel{out(i,j,m)}{\rightarrow} (\varepsilon \circ in(i,j,m)) \circ B\,,$$
$$A \circ B \stackrel{out(k,l,n)}{\rightarrow} A \circ (\varepsilon \circ in(k,l,n))\,.$$

The first possibility follows from the deduction rules as follows. First, $A \stackrel{out(i,j,m)}{\rightarrow} \varepsilon \circ in(i,j,m)$ and second $B \stackrel{out(i,j,m)}{\nrightarrow}$. The second possibility is obtained from $A \stackrel{out(k,l,n)}{\cdots\rightarrow} A$ and $B \stackrel{out(k,l,n)}{\rightarrow} \varepsilon \circ in(k,l,n)$.

An example in which the MSCs to be composed vertically have instances in common is given in Fig. 8. The MSCs D and E are represented by

$$D = out(i,j,m) \circ^{out(i,j,m) \mapsto in(i,j,m)} in(i,j,m)\,,$$
$$E = out(j,k,n) \circ^{out(j,k,n) \mapsto in(j,k,n)} in(j,k,n)\,.$$

The vertical composition of the MSCs D and E is denoted by $D \circ E$. In this case the only initial transition is $D \circ E \stackrel{out(i,j,m)}{\rightarrow} (\varepsilon \circ in(i,j,m)) \circ E$. The execution of $in(i,j,m)$ is blocked by the requirement that $out(i,j,m)$ must be executed first, the execution of $out(j,k,n)$ is blocked by the requirement that all events

on the same instance, i.e., instance j from D, must be executed first (and that is not the case) and the execution of $in(j,k,n)$ is blocked by the requirement that $out(j,k,n)$ must be executed first. Note that there is actually only one sequence of transitions in this example:

$$\begin{aligned} D \circ E &\stackrel{out(i,j,m)}{\rightarrow} (\varepsilon \circ in(i,j,m)) \circ E \\ &\stackrel{in(i,j,m)}{\rightarrow} (\varepsilon \circ \varepsilon) \circ E \\ &\stackrel{out(j,k,n)}{\rightarrow} (\varepsilon \circ \varepsilon) \circ (\varepsilon \circ in(j,k,n)) \\ &\stackrel{in(j,k,n)}{\rightarrow} (\varepsilon \circ \varepsilon) \circ (\varepsilon \circ \varepsilon)\downarrow \ . \end{aligned}$$

4.5 Use of operational semantics

By means of the operational semantics a transition graph can be associated with every expression. Such a transition graph consists of nodes and arrows between those. A node is labelled by an expression over the signature. Suppose that we are given a node s and suppose that we can derive $s \stackrel{a}{\rightarrow} s'$ for certain a and s'. Then the transition graph also contains a node labelled s' and an arrow labelled with a from node s to node s'. This way the transition graph can be built. The transition graph for an expression s has exactly one initial node. This is the node labelled with s. This node is indicated by an incoming unconnected and unlabelled arrow. If for a node s we can derive $s\downarrow$ then this is indicated in the transition graph by labelling that node by an outgoing unconnected and unlabelled arrow.

For example the transition graph associated with $A \circ B$, i.e., the vertical composition of the MSCs A and B from Fig. 7 is given in Fig. 9. Usually we omit the labels of the nodes.

The operational semantics presented can be used to define a notion of equivalence on processes. Examples thereof are *trace equivalence*, *bisimulation equivalence* and *graph isomorphism*. The intended equivalence for MSC'96 is bisimulation semantics. For a definition of this equivalence we refer to [26]. In the case of MSC'96, where we only have deterministic processes, i.e., it is not possible for a process to perform an a-transition to two states represented by different expressions, the notions of trace equivalence and bisimulation equivalence coincide [8].

The reason to use bisimulation equivalence anyway is that we anticipate at an extension of the set of operators with an operator for non-deterministic choice [4]. In the presence of non-deterministic choice, there is a difference between trace semantics and bisimulation semantics. The definition of the

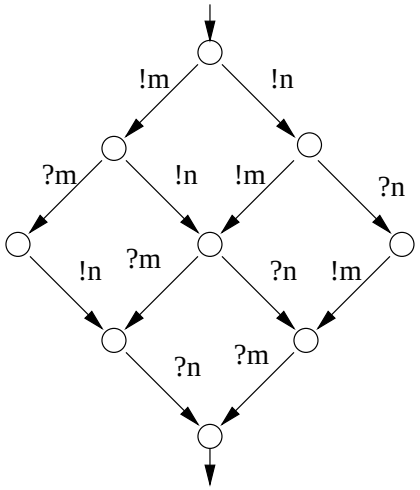

Fig. 9. Transition graph.

operational semantics for MSC'96 is such that it can easily be extended with non-deterministic choice.

Another interesting topic related to operational semantics is the development of a simulator. With a simulator sequences of events can be generated at random or user-driven. The basic functionality of a simulator resembles the definition of an operational semantics. For Basic Message Sequence Charts, i.e., Message Sequence Charts with only instances, messages and local actions, in [21] the process algebra semantics of MSC'93 is used to define a prototype simulator.

5 Summary and concluding remarks

We have explained the basics of the formal semantics of MSC'96 which are currently under development.

The semantics of an MSC is derived in several steps. First, an MSC in graphical representation is translated into a textual form. This transformation is not discussed. Next, the textual representation is translated into a process algebra expression. Finally a meaning is attached to such an expression.

A major design issue was compositionality. An MSC is thought of as being constructed from a number of building blocks, each of which may also be compound constructs. The semantics of a construction is defined as the composition of the semantics of its building blocks. At the lowest level there are only events, which have a trivial interpretation as atomic actions. The ordering of these events is taken care of by the operators for vertical and horizontal composition. Constructs for design in the large are interpreted as applications

of several operators.

We obtained an operational semantics, which consists of a description of the (possible) behaviour of an MSC. We did not provide for any process algebraic axioms as presented in [22,23]. We expect a complete axiomatisation to be infeasible. Nevertheless, an operational semantics as proposed here, serves several purposes. First, it unambiguously defines the meaning of an MSC by interpreting an MSC in the mathematical model of transition graphs. Next, it allows for a good comparison to alternative semantics definitions of MSC, such as approaches based on Petri nets [9], Büchi automata [17], process algebra [19,25], and partial orders [1]. Moreover, it enables a comparison to other languages for the description of distributed systems, such as SDL [12] and LOTOS [7], which are also provided with an operational semantics. Finally, an operational semantics is useful for the development of a simulation tool.

The semantics obtained is mostly an extension of the formal semantics of MSC'93 in [15]. A major difference is that the state operator, used for enforcing the message orderings, is replaced by the generalised weak sequential composition operator. The reason is that the latter allows for a uniform treatment of the message orderings and causal orderings and corresponds closely to the vertical composition of MSCs.

Acknowledgement

We would like to thank the members of the MSC development group for their initiating work. André Engels is acknowledged for the fruitful discussions on some technical matters.

References

[1] R. Alur, G.J. Holzmann, and D. Peled. An analyzer for Message Sequence Charts. *Software - Concepts and Tools*, 17(2):70–77, 1996.

[2] J.C.M. Baeten and S. Mauw. Delayed choice: an operator for joining Message Sequence Charts. In D. Hogrefe and S. Leue, editors, *Formal Description Techniques VII*, IFIP Transactions C, Proceedings 7^{th} International Conference on Formal Description Techniques, pages 340–354. Chapman-Hall, 1994.

[3] J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 149–268. Oxford University Press, 1995.

[4] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[5] J.A. Bergstra, I. Bethke, and A. Ponse. Process algebra with iteration and nesting. *The Computer Journal*, 37(4):243–258, 1994.

[6] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.

[7] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1988.

[8] J. Engelfriet. Determinacy → (observation equivalence = trace equivalence). *Theoretical Computer Science*, 36(1):21–25, 1985.

[9] J. Grabowski, P. Graubmann, and E. Rudolph. Towards a Petri net based semantics definition for Message Sequence Charts. In O. Færgemand and A. Sarma, editors, *SDL'93 - Using Objects*, Proceedings of the Sixth SDL Forum, pages 179–190, Darmstadt, 1993. Amsterdam, North-Holland.

[10] Ø. Haugen. MSC structural concepts. Technical Report TD 9006, ITU-T Experts Meeting SG 10, Turin, 1994.

[11] Ø. Haugen and Y. Lahav. MSC/SDL new features. Tutorials of the Eighth SDL Forum, 1997.

[12] ITU-T. *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*. ITU-T, Geneva, 1988.

[13] ITU-TS. *ITU-TS Recommendation Z.100 Annex I: SDL Methodology Guidelines*. ITU-TS, Geneva, 1993.

[14] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, September 1993.

[15] ITU-TS. *ITU-TS Recommendation Z.120 Annex B: Algebraic semantics of Message Sequence Charts*. ITU-TS, Geneva, April 1995.

[16] ITU-TS. *ITU-TS Draft Recommendation Z.120: Message Sequence Chart (MSC96)*. ITU-TS, Geneva, 1996.

[17] P.B. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

[18] S. Loidl, E. Rudolph, and U. Rinkel. MSC'96 and beyond - a critical look. In A. Cavalli and A. Sarma, editors, *SDL'97: Time for Testing - SDL, MSC and Trends*, Proceedings of the Eighth SDL Forum, pages 213–227, Evry, France, 1997. Elsevier Science Publishers.

[19] J. de Man. Towards a formal semantics of Message Sequence Charts. In O. Færgemand and A. Sarma, editors, *SDL'93 : Using Objects*, Proceedings of the Sixth SDL Forum, pages 157–165, Darmstadt, 1993. Amsterdam, North-Holland.

[20] S. Mauw. The formalization of Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1643–1657, 1996. Special issue on SDL and MSC, guest editor Ø. Haugen.

[21] S. Mauw and E.A. van der Meulen. Generating tools for Message Sequence Charts. In R. Bræk and A. Sarma, editors, *SDL'95 with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 51–62, Oslo, 1995. Amsterdam, North-Holland.

[22] S. Mauw and M.A. Reniers. An algebraic semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.

[23] S. Mauw and M.A. Reniers. An algebraic semantics of Message Sequence Charts. Technical Report CSN 94/23, Eindhoven University of Technology, Department of Computing Science, Eindhoven, 1994.

[24] S. Mauw and M.A. Reniers. High-Level Message Sequence Charts. In A. Cavalli and A. Sarma, editors, *SDL'97: Time for Testing - SDL, MSC and Trends*, Proceedings of the Eighth SDL Forum, pages 291–306, Evry, France, 1997. Elsevier Science Publishers.

[25] S. Mauw, M. van Wijk, and T. Winter. A formal semantics of synchronous Interworkings. In O. Færgemand and A. Sarma, editors, *SDL'93 - Using Objects*, Proceedings of the Sixth SDL Forum, pages 167–178, Darmstadt, 1993. Amsterdam, North-Holland.

[26] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981. Proceedings of the Fifth GI-Conference, Karlsruhe, West Germany.

[27] A. Rensink and H. Wehrheim. Weak sequential composition in process algebras. In B. Jonsson and J. Parrow, editors, *CONCUR'94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 226–241, Uppsala, 1994. Springer-Verlag.

[28] E. Rudolph, J. Grabowski, and P. Graubmann. Tutorial on Message Sequence Charts (MSC'96). Tutorials of the First joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'96), 1996.

[29] J.L.M. Vrancken. The algebra of communicating processes with empty step. *Theoretical Computer Science*, 177(2):287–328, 1997.